EXHIBIT 5

Hello, everybody.
My name is Gokul and together with my colleagues Dima and Jonathan, we are really excited to tell you all about Sign In with Apple.
I will start with an overview of the key features of Sign In with Apple, then Dima will talk through how you can integrate this into your native app.
Jonathan will demo that live here.
And finally, we'll discuss cross-platform and close out with some best practices.
Sign In with Apple is fast, easy account setup and sign in for your application.
It is secure and it's private, both for your users and for your privacy.
It's not Apple's business to know how you engage with your users in your app.
So you can implement Sign In with Apple with confidence that Apple simply won't track any of that.
Let's take a quick look at the experience when your user taps on that button.
They see a sheet pre-filled with information that you request, like name and email.
The user can make a choice on which email they want to share.
They tap Continue and that's it.
They are signed into your application.
Your app gets a unique, stable ID, a name and a verified email address that you can reach for the user.
And best of all, this is a secure two-factor authenticated account for the user in your app.
It's that easy.
And even better, this all works seamlessly across all of the user's devices.
On a new device, a simple tap gets them going with the same account as before in your app and ready to fully engage again.
Very fast, very easy.
So Sign In with Apple provides really streamlined account setup experience for your app.
Notice there are no cumbersome forms to fill out.
Just a simple tap.
Users downloaded your app from the Store using their Apple ID already.
And Sign In with Apple helps them engage fully with just a tap in your app.
You also get a verified email address that you can use to communicate with the user.
Now this is the email address that Apple uses for its communication.
So no need to do these cumbersome round trips of sending email and having users click on links.
Apple has already done that work for you.
Sign In with Apple provides you instantly with an email that just works.

No additional verification required.
Now some users may feel less comfortable sharing their real email address.
And you may end up with some fake emails that don't work.
And Sign In with Apple helps you with that.
Hide My Email option enables these users to share a hidden email address that routes to their verified email inbox.
This is great.
Now this is enabled through Apple's Private Email Relay system that forwards the email to the verified email for the user and it can handle replies too.
So it works both ways.
And you can use this for any kind of business communication for your app.
Apple does not retain any messages once they're delivered to the user's inbox.
In summary, a verified email that works available instantly to your app.
With Sign In with Apple there are no new passwords to remember or forget.
So it's already more secure.
Even better, every account that your app receives is two-factor authentication protected.
This is just the best account for your app.
No password, two-factor authentication, no cost to you and no added friction to the user.
Great built-in security with Sign In with Apple.
Now combatting account fraud in your systems is a difficult problem, and Apple is in a unique position to help.
Sign In with Apple includes a new privacy-friendly way for you to gain confidence that the user on your app is indeed real.
It combines sophisticated on-device intelligence with account information, and abstracts that into a single bit: real user or unknown.
Real users, give them the red carpet treatment in your app they deserve.
This is a high confidence indicator.
If you receive unknown, it could be a user, it could be a bot.
Treat this user like you would treat any other new account in your system where you don't have enough information to make a decision.
So this is the real user indicator helping you fight account fraud.
Finally, Sign In with Apple is cross-platform.
The API is available on all Apple platforms: iOS, MacOS, WatchOS, tvOS.
The sign-in experience is tailored on each platform for ease of use.
The JavaScript API enables you to use Sign In with Apple on the web as well as other platforms like Windows or Android.
We'll talk about this in more detail later.

Streamlined account setup, verified email address, built-in security, anti-fraud and cross-platform.
That's a very quick summary of the key features of Sign In with Apple.
Next, Dima's going to join me onstage to talk through how you can integrate this into your native app.
Dima? Thanks, Gokul.
Hi, folks.
So you've seen this great feature.
What does it actually take to integrate into your app? There are four key things that your app needs to do to hit the ground running.
First, Auto Sign In with Apple-branded button.
Then configure and perform an authorization request.
After the user sees the Sign In with Apple UI and after a quick FaceID check, the results of the authorization will be returned back to your app.
At this point you need to verify the results with the Apple ID servers and create an account in your system.
Last but not least, the credential state may change after it is returned to your app and your application needs to handle those state changes gracefully.
First, let's look into the Sign In with Apple button.
With only a couple lines of code, you can add an AuthorizationAppleIDButton to your app.
Once you initialize it, add an action.
And that's all you need to do.
The button supports several different customizations to fit your app's design.
Different visual styles and different labels are available.
Apps that currently use valid APIs will find this very familiar.
Next, once the user performs the action, you need to configure a request and perform authorization.
Here's how you do it.
With only one line of code, you initialize an Apple ID Authorization request.
This is all your app needs to create an account in your system.
Optionally, if your app requires this for the best user experience, you can set requestedScopes for full name and email.
You should only request this information if it's truly required for your app and err on the side of minimum amount of information.
Once the request is configured, initialize AuthorizationController, set the delegates to get the results back into your app.
And last but not least, perform the request.
Once you perform the request, this will initiate an authorization UI to be presented to the user.
After a quick FaceID check, an authorization result will be returned to your app.

Let's talk about handling these results.
Through an AuthorizationController didCompleteWithAuthorization method, you will get an authorization object.
This object has a credential property of type AppleIDCredential.
You should check that this is actually an AppleIDCredential before processing that.
This object has all the information that is required to create an account in your system.
If the user cancels the request or any other error occurs, we will let your app know through the didCompleteWithError callback.
Both of these delegate callbacks are guaranteed to be made on your app's main queue.
So let's dive deeper into the results that we will give to your app.
First, the user identifier.
This is a unique, stable, team-scoped user identifier.
This is great.
You can use it to retrieve information from your user systems across different platforms, different systems, the web, Android.
It remains stable across all of them.
And it is associated with your developer team.
This is the key to your user.
Next, we will return verification data, an identity token and authorization code.
A short-lived token that you can use with Apple ID servers to exchange for a refresh token.
If you integrate with existing auth systems, adapting Sign In with Apple will be very familiar.
Optionally, we will return account information if you requested it, a name and a verified email.
And since the email is verified by Apple, your apps no longer need to perform any sort of verification after we have shown it to you.
Lastly, a real user indicator, as was mentioned earlier.
Use this to streamline your sign-in experience.
So now you've created an account in your system.
As the user uses Apple devices in your app, the credential state may change.
And you need to handle those scenarios gracefully.
The user may stop using Apple ID with an app.
They may sign out of the device.
Events like this should be handled gracefully.
To allow this, authentication services framework exposes a fast API to allow you to query this state.
Using the user identifier that was previously returned through an Apple ID Credential, you can check GetCredentialState call to get the state of the current Apple ID Credential.

It's a very fast API call.
It can return three things.
We may tell you the user is authorized, and you should let them continue using your app.
The credential may be revoked.
You should sign the user out of your app on this device and optionally guide them to sign back in again.
NotFound means that the user has not previously established the relationship through Sign In with Apple.
This API is very fast.
You should call it on your app's launch to make sure that you provide a tailored experience for each of these states.
Additionally, we expose a notification through NotificationCenter to let you know when this credential state changed to revoked.
Sign the user out on this device and optionally guide them to sign back in again.
Now the user has established the relationship with your app, they're using the app, they create an account.
Inevitably they'll get another device and they'll want to use your app again.
Let's talk about that and how Sign In with Apple helps you there.
When a user comes back to your app on a new device, they will get a one tap sign in experience.
After a quick FaceID check, they're back into your app.
In addition to that, we know you have existing accounts in your systems.
So now we support iCloud Keychain passwords through the same API.
You should request authentication with both Apple ID and iCloud Keychain requests.
The user will be offered to continue using their existing credential, whichever they have.
And if an error is returned, show your regular sign-in flows.
For best user experience, your app should call into this startup function if it does not have an existing local account.
So let's look at how easy it is to implement this.
First, initialize a request array with an Apple ID authorization request and an Apple ID password request.
That's it.
It's that simple.
Pass the request array and perform the request.
If an existing Sign In with Apple connection exists, an AppleIDCredential will be returned to your app.
If the user has a stored iCloud Keychain password, that credential will be returned to your app.
You should use the credential that you get from this API to sign the user in.
If the user has no existing credentials, the API will return an error immediately and you should show your standard login flows.
By performing these requests on startup, you will prevent account duplication within your systems and streamline the user experience

of getting back into your app and start to use it.
Now I'd like to call Jonathan onstage to demo how easy it is to build this.
Thanks, Dima.
Good morning, everyone.
Sign In with Apple simplifies how users create an account or sign into an app while maintaining strong account security.
Password autofill and automatic strong passwords enables you to do some of this already.
Sign In with Apple improves upon them further and you can integrate with password autofill passwords using the same native API as Sign In with Apple.
I'm going to take you through three topics to get your users signed into your app using their Apple ID.
First, I'll add the Sign In with Apple button to a login form.
Then I'm going to show you how to implement the Quick Sign In flow which allows users to sign into existing credentials using just one tap.
Last, I'll show you how to check the credential status of a user identifier after app launch to ensure your users are properly signed in.
To show you all this, I have a test app.
It's called Juice.
Here you can see a typical login form with the email and password.
We're going to add another option below it, Sign In with Apple.
So let's get going.
We have one prerequisite to fulfill in our project settings and that's to add the Sign In with Apple capability.
So in your project settings, you'll want to select your application and go to the Sign In and Capabilities section.
I've already added the Sign In with Apple capability here.
To add it in your app, go ahead and click on the Capability button and then search for Sign In with Apple.
When you do this, Xcode is going to add the appropriate entry into your app's entitlement file.
This will then check that you've added the Sign In with Apple capability to your Apple ID in the Developer Portal.
If you haven't done so already, you'll want to go ahead and sign into the Developer Portal and make sure that you've added this capability.
So while here, I'd like to highlight the Associated Domains Capability seen over here.
This will make sure that you get passwords back in your credential and presented to the user when you make an ASAuthorizationPassword request.
I'll show you how to add one of those requests later in the demo.
If you want your password request to provide exactly the right credentials, refer to the 2017 session Password Autofill for Apps, as well

as Thursday's session What's New in Authentication.
So I'm going to head back to Xcode and we're going to take a look at the Juice app.
As you can see, this is just a simple demo app to highlight some of the key features of this API.
Below this or label I have a Stack View that is called Login Provider Stack View.
This is where we're going to add the Sign In with Apple button.
Adding the Sign In with Apple button will be accomplished in three steps.
First, I'll add the Sign In with Apple button to our Provider Stack View.
Then I'm going to define a function that will create and perform the request when the button is tapped.
And then last, we'll adopt the necessary authentication services protocols which will provide us with results.
So let's get started by adding the Sign In with Apple button to our UI and to do that let's take a look at the LoginViewController Swift file.
So we'll be working with the AuthenticationServices module, so we'll go ahead and import that here.
So here I've defined a function called setupProviderLoginView, and it's initializing the ASAuthorizationAppleIDButton.
It's then adding the handleAuthorizationAppleID ButtonPress function that I've stubbed out over here as its action.
And then I add the button to the loginProviderStackView which will then show it in our UI.
Just above, I'm overriding the viewDidLoad function so that this way we'll go ahead and call setupProviderLoginView and that will get added.
So I'll go ahead and run the app.
And great, the button is now visible in our UI.
But it's not really doing anything, so let's jump to step two which is define the action for our button.
All right, let's talk about what this function is doing.
So what we're doing is we're initializing an ASAuthorizationAppleID request using the AppleIDProvider and then we're setting its requestedScopes property to an array that contains the full name and email scope.
This part is critical.
The user will be asked to share information depending on what scopes you set here.
The information is then returned to your Apple ID Credential -- in your Apple ID Credential.
You should only request what you actually need.
So let's go back to our button's action.
So we initialize a controller using the request that I've inserted into an array, and then we set the delegate and presentationContextProvider to self.
We then call performRequest.
So now that we've defined the action, let's go ahead and run the app.

Tap the Sign In with Apple button.
And as you can see, the UI is asking for the requested scopes which we set in our request.
I'm going to cancel for now.
So remember this delegate in the presentationContextProvider? We're going to have to formally adopt the authenticationServices protocol that's required for this.
I stubbed it out below.
So the ASAuthorizationController delegate has two functions: one that returns an authorization and one that returns an error.
Let's go ahead and we'll define AuthorizationController didCompleteWithAuthorization.
So what we're doing here is we're checking the authorization that comes back.
We'll check the credential of the authorization.
And then in the case in which it's an appleIdCredential, we're going to set a constant user identifier to the user property of the credential.
For demo purposes, we then save the user identifier in the keychain so we can use it later.
I'll also have the ResultViewController that will go ahead and take care of some of the boilerplate of handling some of the properties of the appleIdCredential.
So just below this, just below what we've defined, we have the didCompleteWithError function and this will get called in the case in which the flow doesn't complete or there is no credentials.
Lastly, we have one more protocol and it's ASAuthorizationController PresentationContextProviding.
This defines a function presentationAnchor for controller.
And what this requires us to do is return the window in which we'd like Authentication Services to display the UI.
So let's run the app.
Tap the button and share email.
And authenticate with just one tap.
So the service in here seems to be giving me a bit of a problem, but I've got a backup video that will show you what is supposed to happen.
So here we are.
We are sharing the email and then the ResultViewController will then display the results right there.
And as you can see, we have a user identifier, the full name and email address.
Okay, well, actually it looks like this has worked.
Okay, there we go.
All right, so let's get back on topic.

So to recap, we've added a Sign In with Apple button to our UI, we defined the button's action and the required functions of the authentication services protocol that return results and specify where UI should be shown.
So now that we've covered account creation with the Sign In with Apple button, let's work on getting our users signed in quickly.
To do this, we'll define a function that will display UI to our users in a case in which they have an existing credential.
So this can be either an AppleID or an iCloud Keychain password using the app's authorized domains.
So here I'm defining the performExistingAccountSetup Flows function.
So here we have an array that contains both an Apple ID request and a password request using the respective providers.
We then initialize an authorization controller using those requests, set the delegate and the presentationContextProvider like we did before, and then call performRequest.
So we've added one more type of credential request to our controller and that is the password request.
We will want to handle this type of result in our delegate, so let's go ahead and do that now.
So we're setting a case for the ASPasswordCredential.
So in this case, the ASPasswordCredential has a user and a password property.
In this case, when it gets returned to you, you'll want to go ahead and authenticate against your servers and you don't even need to show any UI to your users.
So there's one last step and that's to actually call the performExistingAccountSetup Flows function.
I'm going to head back up to the class.
I'm going to override viewDidAppear to call performExistingAccountSetup Flows.
Let's run the app.
And as you can see, the user is presented with existing credentials when we show the loginViewController.
That's our QuickSigninFlow.
So we've just implemented the QuickSigninFlow, so let's move on to our last topic which is checking the credential status of an AppleID user identifier.
Earlier in this demo we saved the user identifier to the keychain during account creation.
Using this identifier, we can check the state of the sign in to ensure that our user is properly authenticated.
We'll do this in our AppDelegate.
And I head over to the AppDelegate file.
And I'm going to remove this.
So first we're checking to see if the keychain has the user identifier in it.
If it does, we're going to initialize ASAuthorizationAppleIDProvider, and then using the AppleIDProvider we'll call credentials a state passing the user identifier.

So we're going to get three different -- we have three different types of cases.
One is authorized and in this case you can just assume that the user is properly authenticated and you can continue your normal app operations.
In the case in which it's revoked, you want to call your existing sign out logic and then you can fall through to notFound case which is to show the LoginViewController.
So I'm going to go ahead and run the app.
And as you can see, the user identifier is presented in the ResultViewController because in this case the user is already authorized.
The ResultViewController is the initial view controller in the storyboard.
So that's it.
Pretty easy, right? So I've just shown you how to add the Sign In with Apple button to your login form, how to implement a QuickSigninFlow to get users signed in with an existing credential, and then how to check the credential status using the user identifier to make sure users are authenticated with your app.
With that, I'll hand it back over to Gokul who'll cover multiplatform topics.
Gokul? Thank you, Jonathan.
We just saw Jonathan start with a test app, add a button, handle requests and responses and finally handle existing accounts.
All easy to implement and a great experience for users.
So that was a very quick demo.
Let's talk about cross-platform.
Cross-platform's important and it's enabled through a simple JavaScript library.
Using this library, you can enable your users to sign in with Apple on any platform like Windows or Android.
And clicking the familiar Sign In with Apple button will redirect to Apple where your user can enter the Apple ID and sign in.
Once they sign in, it redirects back.
Both the calls and information you receive back are very similar to the native API.
You get the ID, the token or even the name or email if you request it.
And once you get back the ID and the token, you can convert this to an app session in your app.
Best of all, support for this is built right into Safari.
So when your user clicks that button, Safari will bring up a native Apple Pay-like sheet.
Your user can simply TouchID and they're instantly signed in and using your website super-fast, and what a great experience.
So it's built right into Safari.
Integrating with a JavaScript library is just four easy steps.
You start with including the JavaScript library in your HTML as shown.

A simple div renders the button.
You can style it with many parameters to customize the fit to your site.
Configure it with parameters like whether you want name, email and your redirect URI.
And finally, when the user completes sign in, the results are posted back to your redirect URI with form-encoded values.
You validate the token, the auth code and convert it to an application session.
And you decide how long you want the session to live.
So that's a very quick look at cross-platform support with JavaScript library.
Finally, let's just go through some best practices to observe when integrating with Sign In with Apple.
Here are some general guidelines to follow.
As stated in the App Store guidelines, unless your app requires significant account-based features, let people use it without a login.
For example, you can guide the user to sign in with Apple after they've made a purchase so you can tie their purchase to an account that they can easily get back to later.
If you just need a unique identifier to identify the user, don't collect name or email.
You don't need it.
And if you do collect email through Sign In with Apple, make sure that you respect the user's choice.
And here are some additional best practices to keep in mind when integrating with the API.
When your app first starts up, use the API to check for existing accounts.
This allows the user to quickly get going with an account they already have with your app, whether it's iCloud Keychain password or an existing Apple account.
And you don't have any duplicate accounts in your app.
Plan to provide the best experience for users that return real user indicator.
If it returns unknown, treat them like you would any new account in your system.
Use the Button API to draw the button.
And when you draw the button with the API, wire it to the Apple ID Provider in your code.
And finally, once users use Sign In with Apple, they will expect to see it on all platforms that your app is on.
So implement it across all of your platforms.
So that's a quick look at best practices.
In summary, Sign In with Apple is fast, easy account setup and sign in for your app.
Streamlined, one-tap account setup with no cumbersome forms.
Verified email address that instantly works and you can use for any communication.
Built-in security with no new passwords and two-factor authentication for every account.

Real user indicator to help you combat account fraud.
And cross-platform support so your users can benefit from Sign In with Apple on every platform that your app is on.
Please join us at our lab at 10:00 where we're happy to answer any questions you may have.
There's more in the authentication services framework to help you help your users with signing in and dealing with their passwords.
See the What's New in Authentication session tomorrow to learn more.
And Sign In with Apple works great with independent Watch apps.
Please tune in to Independent Watch Apps on the WWDC app to learn more.
That's all.
Thank you for joining us.
Have a wonderful conference.
Thank you.